

by  
Jeff Proise

# Lab Notes

Even inexpensive modems run at 2,400 bits per second today, and newer designs push the upper limit to 9,600 bps and beyond. Direct serial links between machines allow even higher transmission rates. 300-bps communications programs went out with the Hula Hoop—about the time the designers of DOS wrote its serial driver.

There's nothing in DOS itself to preclude high-speed serial I/O, of course. It's just that a DOS communications program has to write its way around the operating system and supply its own communications facilities. In this first of a two-part series on serial communications programming, we give an overview of the DOS communications environment, then present and analyze a terminal emulation program with advanced input capabilities. Next time, we'll do the same for the more hospitable OS/2 communications environment. Part 2 will culminate in the code listing for a protected-mode terminal emulator that highlights the differences between the two environments and permits a machine running OS/2 to communicate with another running DOS.

The communications resources that DOS supplies are meager at best. Only one serial port can be accessed at a time, and the DOS AUX serial device driver is slow, rudimentary in design, and adequate only for polled I/O. In a polled I/O system the speed must be slow enough to allow the CPU to spend an appreciable fraction of its clock cycles checking each port in turn ("polling") to see whether a character is waiting to be either transmitted or received. This is impractical for speeds greater than 300 bps.

Handling high-speed data transfers requires more-sophisticated, interrupt-driven I/O techniques. In an interrupt-driven system the CPU doesn't waste clock cycles looking for work. Instead, the UART chip that drives the serial port grabs the attention of the CPU ("interrupts") only when a character actually arrives. If the interrupt handler is written with sufficiently

**HIGH-SPEED COMMUNICATIONS IN DOS AND OS/2: Under DOS you need special programming techniques to implement interrupt-driven operation. We'll take a look at these and, in the next issue, explore OS/2's support for serial communications.**

low overhead, the chance of losing an incoming character because the last character is still being processed is virtually nil. Similarly, in interrupt-driven output, the serial port notifies the CPU when another character can be transmitted, freeing it to keep up with other tasks in the interim.

The DOS services that support serial I/O through the interrupt 0x21 interface are simply inadequate for interrupt-driven applications. (Note: The 0x prefix is the C language method of denoting hexadecimal numbers, a convention we've adopted here.) Indeed, since DOS provides no facilities for programs even to initialize the serial port to given transfer rates and line control settings, programmers must fall back either on using the ROM BIOS or programming the serial port hardware directly. As we'll see, however, the BIOS

offers only slightly better support for serial I/O than does DOS itself. Thus, as a practical fact of life, DOS programmers have long grown accustomed to bypassing the operating system and firmware altogether to transmit and receive serial data.

## DOS COMMUNICATIONS ENVIRONMENT

The heart of a serial port is a chip called the Universal Asynchronous Receiver/Transmitter, or UART. It is the UART that physically transmits and receives characters through the pins of an RS-232 connector. When a character is to be transmitted, the UART frames it with the proper start, stop, and parity bits. The UART then puts this data, one bit at a time, onto the TD (Transmit Data) pin, at predefined intervals. Likewise, when an incoming data packet is received on the RD (Receive Data) pin, the UART disassembles the packet and presents it in the form of an unframed 7- or 8-bit character. The specific UART used in the IBM PC family is a National INS8250 Asynchronous Communications Controller (or a workalike). It's more commonly referred to as "the 8250," or simply "the UART."

The UART also supplies a number of higher-level functions that help put character input and output under program control. If desired, it will compare the parity bit of each character it receives against a parity value it calculates itself and report a *parity error* if the two don't agree. It will also report a *receiver overrun* error if a received character isn't read from the UART's internal buffer before the next one arrives, and a *framing error* if an incorrect stop bit is detected.

Registers inside the UART hard-wired to the RS-232 control pins allow programs to perform hardware *handshaking* with devices connected to the serial port. Thus, for example, a transmitting device may assert the RTS pin, indicating that it is Ready to Send a character. Before actually sending the character, however, the transmitting device may require confirmation that the device on the other end of the line is

ready to receive. The UART provides this confirming "handshake" by asserting the CTS (Clear to Send) pin.

The BIOS does provide four interrupt 0x14 functions designed to permit applications to use the serial port without requiring programmers to deal directly with the inner complexities of the UART. These functions are detailed in Figure 1. Function 0 initializes the UART to the data transfer rate (expressed in bits per second) and line control settings specified by the bit pattern in AL. Rates up to 9,600 bps are supported. Function 1 transmits the character in AL to the serial port, while function 2 reads a character from it. Function 3 returns a 16-bit value that represents the status of the serial port in AX. On return, AH reflects the current line status (acquired directly from the UART's Line Status register) and AL holds the modem status bits (obtained from the Modem Status register).

Functions 2 and 3 use simple hardware handshaking schemes to make sure that spurious data is neither transmitted nor received. Function 2 asserts DTR (Data Terminal Ready) and RTS (Ready to Send) before outputting a character. It then waits for DSR (Data Set Ready) and CTS (Clear to Send) to be asserted in response. Function 3 asserts DTR, then waits for DSR to be asserted before attempting to read a character.

The vital shortcoming of these services, however, is that the BIOS has no way to alter the handshaking modes. The BIOS services are thus generally limited to use in two programs designed to talk to each other, or to situations in which the device on the other end of the line is known to employ the same protocol.

Figure 2 shows how several DOS interrupt 0x21 services use the BIOS to provide simple input and output services. Function 3 of interrupt 0x21 waits for a character to appear at the serial port and returns it in AL. Function 4 transmits the character in DL. The primary difference between the DOS and BIOS character transmit and receive services is that the BIOS routines will return to the caller with bit 7 of AH set if a time-out counter expires before a character can be transmitted or read. The DOS functions, by contrast, will hang in an endless loop. You can prevent such hangups by checking the status of the UART before



## BYPASSING THE UART

### Function 0: Initialize Serial Port

Call with: AH = 0  
AL = Communications parameters (see below)  
DX = Serial port number (0-3)

Returns: AH = Line status (see function 3 below)  
AL = Modem status (see function 3 below)  
Bit 7 is set in AH if function timed-out

Bit settings in the communications parameters byte:

| Bits 7, 6, 5 | Data rate | Bits 4, 3 | Parity setting      |
|--------------|-----------|-----------|---------------------|
| 000          | 110 bps   | X0        | No parity           |
| 001          | 150 bps   | 01        | Odd                 |
| 010          | 300 bps   | 11        | Even                |
| 011          | 600 bps   | Bit 2     | Number of stop bits |
| 100          | 1,200 bps | 0         | 1 bit               |
| 101          | 2,400 bps | 1         | 2 bits              |
| 110          | 4,800 bps | Bits 1, 0 | Number of data bits |
| 111          | 9,600 bps | 10        | 7 bits              |
|              |           | 11        | 8 bits              |

### Function 1: Output Character

Call with: AH = 1  
AL = Character code  
DX = Serial port number (0-3)

Returns: AH = Line status (see function 3 below)  
Bit 7 is set in AH if function timed-out

### Function 2: Input Character

Call with: AH = 2  
DX = Serial port number (0-3)

Returns: AH = Line status (see function 3 below)  
AL = Character code  
Bit 7 is set in AH if function timed-out

### Function 3: Get Serial Port Status

Call with: AH = 3  
DX = Serial port number (0-3)

Returns:

| AH = Line Status  | AL = Modem Status                          |
|---|--|
| Bit 7 = Undefined   | Bit 7 = DCD (Data Carrier Detect) asserted |
| Bit 6 = Transmit Holding and Transmit Shift registers empty | Bit 6 = RI (Ring Indicator) asserted       |
| Bit 5 = Transmit Holding register empty                     | Bit 5 = DSR (Data Set Ready) asserted      |
| Bit 4 = BREAK detected                                      | Bit 4 = CTS (Clear to Send) asserted       |
| Bit 3 = Framing error                                       | Bit 3 = Delta DCD                          |
| Bit 2 = Parity error  | Bit 2 = Delta RI                           |
| Bit 1 = Overrun error                                       | Bit 1 = Delta DSR                          |
| Bit 0 = Data ready in Receive Buffer register               | Bit 0 = Delta CTS                          |

**Figure 1:** The BIOS interrupt 0x14 does provide programmers with a set of functions for creating applications that can access the serial port, without requiring them to deal directly with the inner complexities of the UART. However, these services lack the flexibility needed to support a variety of handshaking methods and are, therefore, generally limited.

calling function 3 or 4. The only way to do this without bypassing the operating system altogether, however, is by making a separate call to IOCTL, which is a highly inefficient procedure.

There are, as indicated, additional problems about DOS's serial I/O support. For one, the only means DOS provides to initialize a serial port or point the AUX driver to a device other than COM1 is by execution of a MODE command from the command line. Worse, like the BIOS services, the DOS services are inherently suited for polled I/O only; interrupt-driven operation requires special programming.

## FACING THE INEVITABLE

Thus, when running under DOS, neither its services nor the BIOS's are adequate for high-speed serial communications. The functionality needed for interrupt-driven I/O is available only by dispensing with the operating system's aid and working directly at the UART level.

The UART is programmed to generate an interrupt when an I/O-related event occurs, and the 8259 Programmable Interrupt Controller is programmed to pass the interrupt on to the CPU. By setting selected bits in its Interrupt Enable register, the UART can be made to generate an interrupt under a wide variety of conditions: when a byte is received in its Receive Buffer register; when its Transmit Holding register is ready to accept another character; when a parity, overrun, or framing error occurs; when a BREAK is detected; or when the state of an RS-232 input pin

changes. The application program sets up interrupt handlers to service each interrupt it has chosen to recognize. If more than one type of interrupt is enabled, the program can determine which condition triggered the most recent one by reading the UART's Interrupt Identification register.

A layout of the UART registers is shown in Figure 3. Within the PC, a table containing the 16-bit I/O addresses of all UARTs installed in the system is stored at offset 0 in the BIOS data area. Once a UART's base address is obtained, its internal registers are accessed relative to that base. An address of 0 indicates that the corresponding serial port is not installed. Figure 4 shows how the UART generates and identifies interrupts.

In addition to the Interrupt Enable and Interrupt Identification registers, the Receive Buffer/Transmit Holding, Line Control, and Modem Control registers also play an important role in interrupt-driven serial communications. The Receive Buffer/Transmit Holding register performs double-duty: a byte written to it with an OUT instruction is transferred to the UART's Transmit Shift register and output; a byte read from it with an IN is pulled from the UART's Receive Buffer register.

The Line Control register contains data format information, which defines the number of data bits, the number of stop bits, and the parity setting. In addition, bit 7 of this register serves as the Divisor Latch Access Bit, or DLAB. When DLAB is zero, the ports at offsets 0 and 1 from the base address correspond to the UART's Receive Buffer/Transmit Holding and Interrupt Enable registers, respectively; when DLAB is 1, the same I/O addresses are mapped to the Baud Rate Divisor registers. An interrupt service routine that reads the serial port must thus take care to clear DLAB before reading a character with an IN from the address at offset 0. Otherwise, the character returned may actually have come from the LSB (Least-Significant-Bit) Baud Rate Divisor register.

Finally, the Modem Status register contains the DTR (Data Terminal Ready) and RTS (Request to Send) control bits. It also contains the general-purpose GPO2 output bit. In the IBM implementation, this bit must be set before UART interrupts can reach the CPU. Bits that correspond to the RS-232 input pins DSR (Data Set Ready), CTS (Clear to Send), RI (Ring Indicator), and DCD (Data Carrier Detect), when needed, are found in the Modem Status register.

## THE DOS COMMUNICATIONS MODEL

The core of a DOS communications program is the set of interrupt service routines (ISRs) that processes UART interrupts. This module may either be incorporated within the program itself, or it may take the form of an installable device driver that is loaded from the CONFIG.SYS file. Most software writers choose the former alternative, reasoning that an application program should be written so that its operation is independent of external resources. Moreover, an I/O module contained in an .EXE file is erased when the program terminates.


**The number and sophistication of interrupt service routines that are written depends upon the nature of the communications services needed.**

A device driver, on the other hand, remains in memory until the system is restarted. This wastes RAM that could be used by subsequently loaded application programs.

The number and relative sophistication of interrupt service routines to be written depends upon the nature of the communications services needed. A simple application, such as the terminal emulator presented here, will have at least one ISR to intercept interrupts triggered when a character is received by the UART. The ISR reads incoming characters and buffers them in a circular first-in, first-out (FIFO) queue.

The balance of the program—the portion that processes input—reads from the queue rather than from the UART. The procedure is identical to the way the BIOS buffers keystrokes in the keyboard buffer and makes them available through interrupt 0x16.

As long as the queue isn't filled, the primary program routines are freed from the time constraints that are normally associat-



### HOW DOS SUPPORTS SERIAL I/O

|                                     |                     |
|-------------------------------------|---------------------|
| <b>Function 3: Input Character</b>  |                     |
| Call with:                          | AH = 3              |
| Returns:                            | AL = Character code |
| <b>Function 4: Output Character</b> |                     |
| Call with:                          | AH = 4              |
|                                     | DL = Character code |
| Returns:                            | Nothing             |

**Figure 2:** The rudimentary serial I/O services that DOS interrupt 0x21 supplies do not include any way of initializing ports and will hang if a time-out error is encountered.

# COREL DRAW! 1.1

## The Best Gets Better

### More Typefaces!

- More than 100 included free
- More than 4,300 can be incorporated from: Adobe, Bitstream, Compugraphic, Casady & Greene, Digifonts, HP Type Director, Image Club, The Font Company, Treacyfaces.

### More Clipart!

- More than 300 are included free (over 6 Mbytes!)
- More than 10,000 compatible images are available from: ArtRight, Casady & Greene, DreamMaker, Dynamic Graphics, Image Club, Marketing Graphics Inc., Metro Image Base, Micromaps, Multi-Ad, New Vision, T-Maker, 3G.

### More Connectivity!

- Top quality slides using Matrix Slidemakers (SCODL)
- Windows Clipboard, CGM, IBM Mainframe graphics.



*The World's Finest  
PC Illustration Package  
Does It Again!*

Corel DRAW has rapidly received acclaim as the most exciting graphics package in the industry - with ease of use, versatility and speed that exceeds even Macintosh programs!

Now version 1.1 rockets even further ahead with unprecedented value and capability. We bundle in a phenomenal amount of free typefaces and clipart images that would normally cost thousands of dollars. In addition, our new type converter enables thousands of additional fonts to be incorporated. Furthermore,

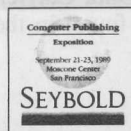
virtually unlimited clipart is available with over 10,000 images supported from 12 leading clipart vendors.

Corel DRAW is the ideal illustration partner to Ventura®, Pagemaker®, WordPerfect®, MS Word®, Ami®, Lotus 123®, Excel® and hosts of other programs.

Corel DRAW 1.1 is shipping NOW and you can upgrade your 1.0 version for \$99. Call us for direct orders or dealer information.

Corel DRAW! 1.1 \$595  
1.1 Upgrade \$ 99

See you  
at Booth  
1942



 **COREL**  
(613) 728-8200

CIRCLE 371 ON READER SERVICE CARD

## Lab Notes

ed with high-speed asynchronous I/O processing. And since the ISR is responsible only for reading and buffering incoming data, unless the transfer rate is extremely high the likelihood that it will miss a character is slight.

More sophisticated ISRs shoulder additional burdens. If high-speed data output is required, an application can effect interrupt-driven output by setting up a separate ISR to service the UART interrupts that are generated as the Transmit Holding register is emptied.

Output will be directed to a holding buffer and it will then be spooled to the UART, a character at a time, by the ISR. Additional ISRs can also handle errors reported by the UART, if desired.

A properly written communications program running under DOS can achieve data transfer rates beyond 9,600 bps even on a 4.77-MHz PC, as is proved by some of the commercial utilities that have been designed to transfer data between laptop

PCs and desktop models over specially built cables. Faster machines can handle even higher rates.

Note, however, that at high transfer rates approaching the threshold of a given PC's capabilities, programs must usually be optimized for speed at the expense of handshaking and parity checking. Under these circumstances, file-transfer protocols with fixed packet sizes are used to make sure that transmission errors are detected and corrected.

## IMPLEMENTING THE MODEL

The schematic representation of a communications program that carries its own interrupt-driven serial input logic on-board is shown in Figure 5. Incoming data is received and buffered by an ISR and the program, as described above, reads data from the input queue. In the typical fashion of a terminal emulator, in which the output rate is no greater than the speed of the typist, outgoing data is delivered directly to the UART. Note that the operating system and

| THE UART REGISTERS |                                 |
|--------------------|---------------------------------|
| Base Address       |                                 |
| +0:                | Receive Buffer/Transmit Holding |
| +1:                | Interrupt Enable                |
| +2:                | Interrupt Identification        |
| +3:                | Line Control                    |
| +4:                | Modem Control                   |
| +5:                | Line Status                     |
| +6:                | Modem Status                    |
| +0:                | LSB Baud Rate Divisor*          |
| +1:                | MSB Baud Rate Divisor*          |

\*Mapped into this address when DLAB = 1. DLAB is the Divisor Latch Access Bit, which is bit 7 of the Line Control register. When DLAB is 0, the Receive Buffer/Transmit Holding register is accessed at offset 0 and Interrupt Enable at offset 1. When DLAB is 1, the same I/O addresses correspond to the Baud Rate Divisor Latch registers.

**Figure 3:** To access and program an internal register of the UART, you must first obtain the correct base address from the table stored in the BIOS data area. To write to the internal registers listed above, you then add the corresponding offset values to the base address. The range of services that each UART register provides is detailed in Figure 4.



## HOW THE UART GENERATES AND IDENTIFIES INTERRUPTS

### Interrupt Enable Register:

|                  |   |
|------------------|---|
| Bit 7 = Always 0 | Bit 3 = Change in RS-232 input pin            |
| Bit 6 = Always 0 | Bit 2 = Error or BREAK detected               |
| Bit 5 = Always 0 | Bit 1 = Transmit Holding register empty       |
| Bit 4 = Always 0 | Bit 0 = Data ready in Receive Buffer register |

### Interrupt Identification Register:

|              |                                       |     |                                 |
|--------------|---------------------------------------|-----|---------------------------------|
| Bits 3, 2, 1 | Interrupt source                      | 010 | Transmit Holding register empty |
| 001          | No interrupt                          |     |                                 |
| 110          | Error or BREAK detected               | 000 | Change in RS-232 input pin      |
| 100          | Data ready in Receive Buffer register |     |                                 |

### Line Control Register:

|                                  |                |           |                     |
|----------------------------------|----------------|-----------|---------------------|
| Bit 7 = Divisor Latch Access Bit |                | Bit 2     | Number of stop bits |
| Bit 6 = BREAK control            |                | 0         | 1 bit               |
| 0                                | Off            | 1         | 2 bits              |
| 1                                | On             |           |                     |
| Bits 5, 4, 3                     | Parity setting | Bits 1, 0 | Number of data bits |
| 000                              | No parity      | 00        | 5 bits              |
| 001                              | Odd            | 01        | 6 bits              |
| 011                              | Even           | 10        | 7 bits              |
| 101                              | Mark           | 11        | 8 bits              |
| 111                              | Space          |           |                     |

### Modem Control Register:

|                       |              |
|-----------------------|--------------|
| Bit 7 = Always 0      | Bit 3 = GPO2 |
| Bit 6 = Always 0      | Bit 2 = GPO3 |
| Bit 5 = Always 0      | Bit 1 = RTS  |
| Bit 4 = Loopback test | Bit 0 = DTR  |

### Line Status Register:

|   |   |
|---|---|
| Bit 7 = Always 0  | Bit 3 = Framing error                         |
| Bit 6 = Transmit Holding and Transmit Shift registers empty | Bit 2 = Parity error                          |
| Bit 5 = Transmit Holding register empty                     | Bit 1 = Overrun error                         |
| Bit 4 = BREAK detected                                      | Bit 0 = Data ready in Receive Buffer register |

### Modem Status Register:

|                      |                   |
|----------------------|-------------------|
| Bit 7 = DCD asserted | Bit 3 = Delta DCD |
| Bit 6 = RI asserted  | Bit 2 = Delta RI  |
| Bit 5 = DSR asserted | Bit 1 = Delta DSR |
| Bit 4 = CTS asserted | Bit 0 = Delta CTS |

**Figure 4:** By programming the UART directly, the full range of services needed for interrupt-driven serial communications is made available.

BIOS are merely spectators; all interaction takes place between the communications program and the serial port hardware.

This schema is implemented in DOSTERM, a simple but fully functional terminal emulation program that permits characters typed at one PC to appear on the screen of another. The COM1 ports of the two PCs should be tied together by a null modem cable—that is, a cable in which the TD and RD lines are cross-wired so that what is transmitted on TD at one end presents itself on RD at the other end.

The assembly language listing for DOSTERM is shown in Figure 6. The program is designed to be assembled and linked into a standalone .EXE file. Following the conceptual model diagrammed in Figure 5, DOSTERM sets aside a 1,024-byte internal buffer for incoming characters and provides an ISR—labeled READ\_COM in the source code—to service them. It displays incoming data on the screen, echoes any characters typed at its own keyboard to the screen before transmitting them out COM1, and terminates when the Esc key is pressed.

In the interest of simplicity, DOSTERM performs no handshaking with the external device to which it is linked. (It does, however, assert DTR and RTS, just in case the device at the other end relies on these pins.) Nor does it offer any means of flow control—a subject that will be treated in more detail in next issue's discussion of OS/2 communications programming.

At startup, DOSTERM initializes the UART and the 8259A for interrupt-driven operation. The appropriate software interrupt—0x0C for COM1 (0x0B would have been used had the program been written for COM2)—is altered to point to an ISR that provides low-level communications services. The ISR is complemented by two routines, READ\_CHAR and SEND\_CHAR, that act as an interface to the rest of the program. It calls these routines as needed to read and write characters to and from the serial port, completely oblivious to the fact that incoming characters are being buffered in the background by another component of the same program. Just before the program terminates, it deprograms the UART and 8259A and restores the interrupt vector that was displaced at runtime. After performing normal housekeeping chores, it ends, leaving

it started.

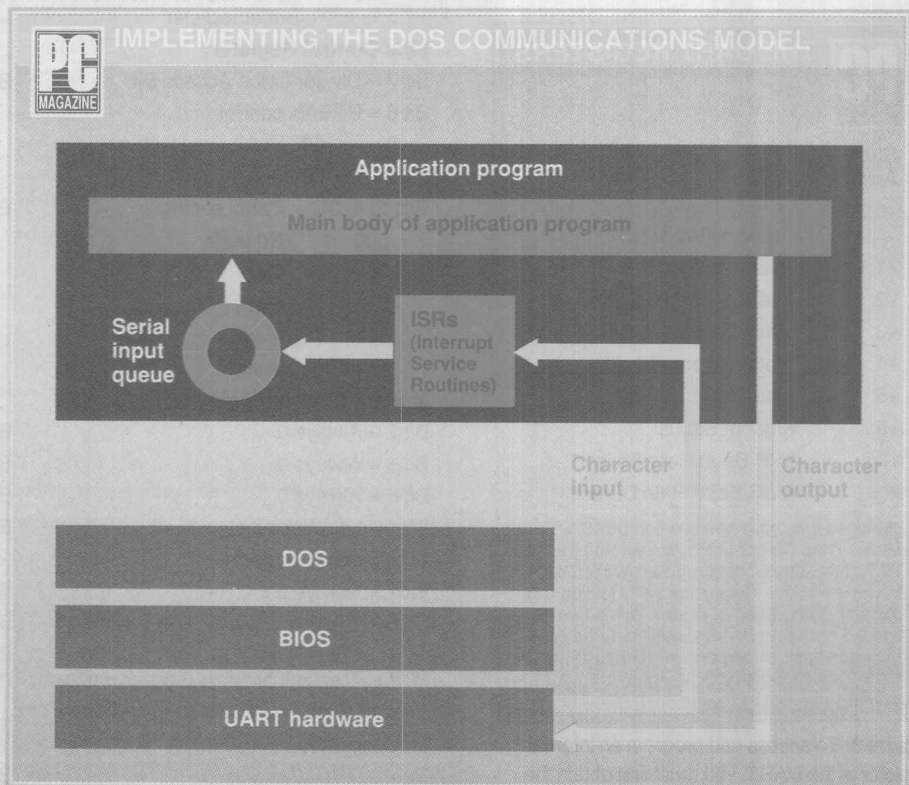
Proper execution of the hardware initialization sequence is critically important. To set up its host for interrupt-driven I/O, DOSTERM performs the following steps:

- Obtains the base address of the COM1 UART from the BIOS data area;
- Changes the interrupt 0x0C vector to route UART interrupts to the interrupt service routine READ\_COM;
- Initializes the UART to 9,600 bps, no parity, 8 data bits, and 1 stop bit, using the BIOS interrupt 0x14, function 0 (the serial port setup service);
- Unmasks IRQ4 interrupts (corresponding to COM1) by clearing bit 4 of the 8259A's interrupt mask register at I/O address 0x21;
- Sets bit 0 of the UART's Interrupt Enable register so that an interrupt will be generated whenever a character is received by COM1;
- Asserts bit 3 of the UART's Modem Control register so interrupts can reach the CPU. At the same time, bits 0 and 1 are set, enabling DTR and RTS, which signi-

DOSTERM takes a shortcut by using the BIOS to initialize the UART to the desired data transfer rate and line settings. As an alternative, it could have written the values directly to the Line Control and Divisor Baud Rate registers.

Interrupts generated by COM1 and COM2 are tied to the PC's dedicated IRQ4 and IRQ3 hardware interrupt lines, respectively. These, in turn, are associated with software interrupts 0x0C and 0x0B. The first step in setting up an ISR to service UART interrupts is to point the appropriate interrupt vector to it. In DOSTERM, which uses COM1 exclusively, this interrupt is 0x0C. The original value of the vector is saved before modification so that it can be restored upon termination.

No hardware interrupt can reach the CPU until the corresponding bit in the 8259A's interrupt mask register is set to 0. Bits 3 and 4 are assigned to IRQ3 and IRQ4, respectively, and the default setting of both bits is 1. This disables both the IRQ3 and IRQ4 interrupts unless a program takes explicit action to enable them.



**Figure 5:** When a DOS communications program requires interrupt-driven input, both the operating system and BIOS must be bypassed and the UART programmed directly. In this model, an ISR handles interrupts triggered by the UART when a character is waiting to be read. The character is buffered in a circular queue from which the program can retrieve it when time permits. Outgoing characters are sent directly to the UART's Transmit Holding register, a technique typical in a terminal emulator whose data transmission rates are low.

# You know exactly what your company wants in a color printer.

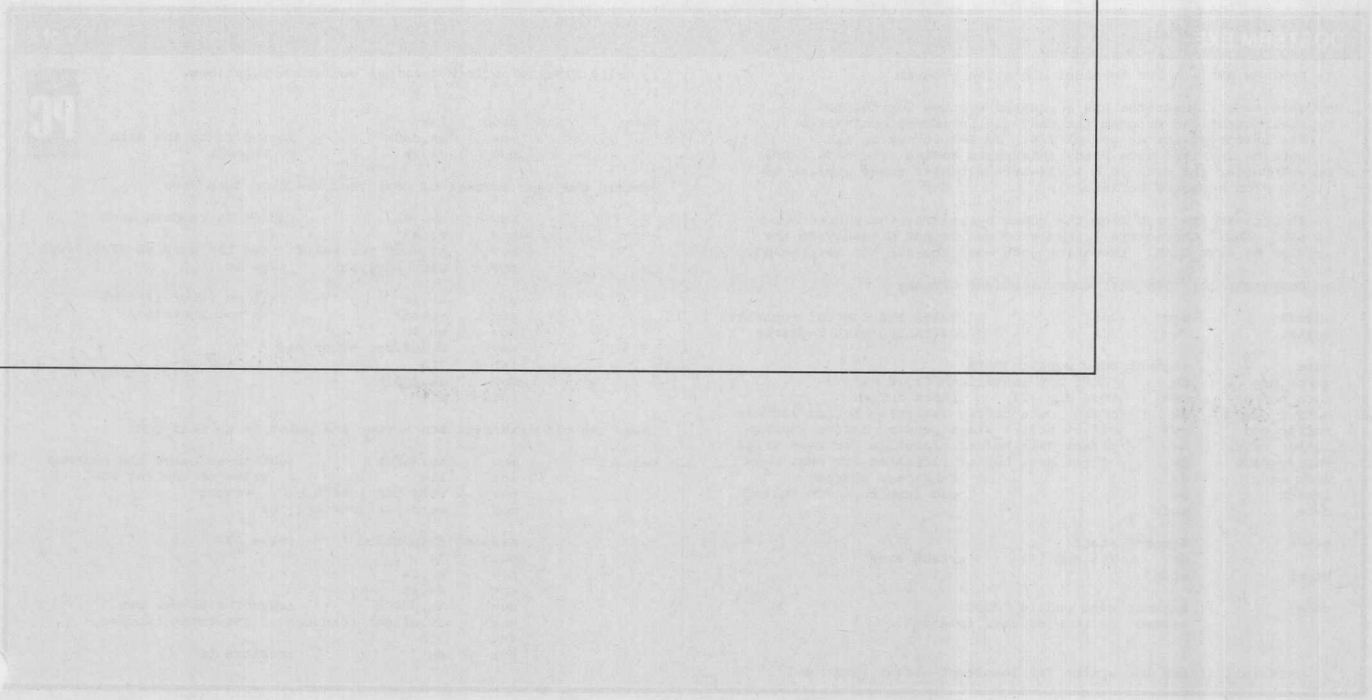


Figure 6: The assembly language listing for C08TERM.EXE, a simple interrupt-driven terminal emulation program that allows characters typed on one PC to be sent, via a null modem cable, to the console of a second PC. C08TERM performs no handshaking with the external device to which it is linked.

## Lab Notes

DOSTERM enables COM1 interrupts but leaves COM2 interrupts masked off. It then follows up by asserting GPO2 (bit 3) in the UART's Modem Control register. GPO2 controls a hardware gateway in the PC that, when closed, cuts off UART interrupts before they reach the 8259A.

At this point, the ISR is being properly addressed by an interrupt vector, and the path from the UART hardware to the CPU is clear. It then remains only to activate UART interrupts by setting bit 0 of its Interrupt Enable register. By default, all bits in this register are clear. With bit 0 set, interrupts are triggered when characters are delivered to the Receive Buffer register, a sign that data is waiting to be read. Other bits control other interrupt options. Had DOSTERM been endowed with interrupt-driven output capabilities, it would have also set bit 1 so that it would be notified when the Transmit Holding register could accept another character for output.

Adding interrupt-driven output would have introduced one further complication, however. Since only one interrupt vector is assigned to each serial port, if two ISRs are needed to service it (one for data-ready interrupts, another for Transmit Holding register-ready interrupts), both ISRs

would have to be combined into a single ISR package. A dispatcher at the front end of the package would be required to determine which condition triggered the interrupt and to route execution to the input or the output ISR appropriately.

The key to performing such electronic gymnastics is the UART's Interrupt Identification register. When interrupts are multiplexed, the lower 3 bits of the Interrupt Identification register identify the source of the last interrupt. A Data Ready condition, for example, is characterized by the value 4. A value of 2 identifies a Transmit Holding register-ready interrupt. Other interrupt ID values are shown in the section of Figure 4 that maps the Interrupt Identification register.

READ\_COM reads the serial port and puts each successive character into a 1,024-byte FIFO queue named DATA\_BUFFER. Two pointers into the buffer—BUFFER\_HEAD and BUFFER\_TAIL—designate where the next character will be written to and extracted from, respectively.

Incoming characters are copied into the location indexed by BUFFER\_HEAD. After a character is inserted, BUFFER\_HEAD is advanced so that the next character will not overwrite the last. When a character is read from the buffer, it is retrieved from the address held in BUFFER\_TAIL, and BUFFER\_TAIL is in

turn incremented to the next position. Thus, when the head and tail addresses are equal, the buffer is empty, and when the next increment of the head would allow it to overtake the tail, the buffer is full. If a new character is received with the buffer full, it is ignored. The next one will be duly buffered provided that a slot has opened for it in the queue.

To ensure that it gets top priority when an interrupt is received, READ\_COM leaves interrupts disabled while it runs. Before it ends, it sends an end-of-interrupt command to the 8259A to signal that processing is completed. Until this occurs, hardware interrupts of equal and lower priority are withheld from the CPU.

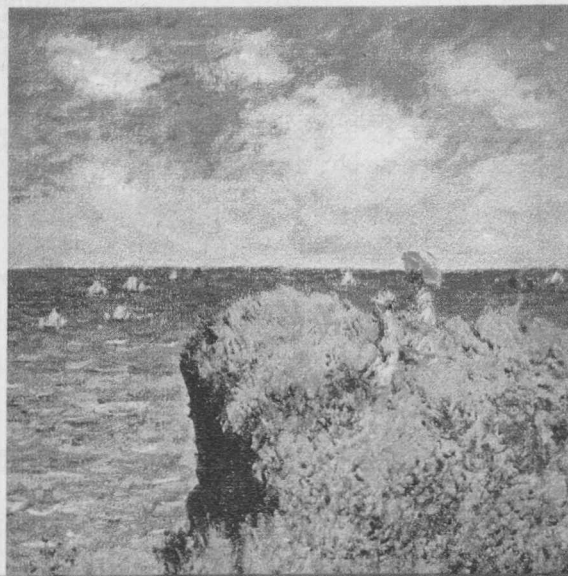
READ\_COM operates asynchronously from the main body of the program, which is contained in the procedure TERM. TERM executes a simple loop, alternately polling the keyboard and serial input buffers for data. When a keycode appears in the keyboard buffer, it is read, displayed, and transmitted; when a character appears in the serial input queue, it is read and displayed. The queue is checked for new characters simply by comparing the head and tail addresses. If the addresses are the same, no data is waiting to be read.

Characters are read from the queue with the subroutine READ\_CHAR. To ensure that it isn't interrupted by READ\_COM while in the midst of extracting a character

| DOSTERM.EXE   |  | 1 of 2 |
|---|--|--------|
| <pre> ;; DOSTERM.ASM - A DOS terminal emulation program ;; ;; This code illustrates how a program written for the DOS ;; environment can program the UART and interrupt controller ;; for interrupt-driven serial I/O. An ISR is set up to ;; service received data ready interrupts coming from IRQ4 (COM1) ;; and buffer the data in a 1,024-byte circular queue similar to ;; the BIOS keyboard buffer. ;; ;; Characters are read from the queue by calling the subroutine ;; READ_CHAR. Characters transmitted are output directly to the ;; UART by SEND_CHAR. Execution ends when the Esc key is pressed. ;; ;; Copyright (c) 1989 Ziff Communications Company  INTA00 equ 20h ;8259A IRQ control register INTA01 equ 21h ;8259A IRQ mask register  data segment word public 'DATA' error_msg db "COM1 not installed",13,10,"\$" data_buffer db 1024 dup (?) ;input buffer buffer_start dw offset data_buffer ;starting buffer address buffer_end dw offset buffer_start ;ending buffer address buffer_head dw offset data_buffer ;location for next write buffer_tail dw offset data_buffer ;location for next read uart_addr dw ? ;UART base address int0Ch dd ? ;old interrupt 0Ch vector data ends  stack segment stack dw 256 dup (?) ;stack area stack ends  code segment word public 'CODE' assume cs:code,ds:data,ss:stack  ;; ;; MAIN initializes the system for interrupt-driven input and </pre> |  |        |
| <pre> ;; calls TERM to perform terminal emulation functions. ;; main proc far mov ax,data ;point DS to the data mov ds,ax ; segment  ; Obtain the base address of COM1 from the BIOS data area. ; mov ax,40h ;point ES to data area mov es,ax mov ax,word ptr es:[0] ;get the word at 0040:0000h mov uart_addr,ax ;save it  or ax,ax ;exit on error if COM1 jnz vector ; is not installed mov ah,9 mov dx,offset error_msg int 21h mov ax,4C01h int 21h  ; Save the old interrupt 0Ch vector and point it to READ_COM. ; vector: mov ax,350Ch ;obtain and save the current int 21h ; value of the int 0Ch mov word ptr int0Ch,bx ; vector mov word ptr int0Ch[2],es  assume ds:nothing ;save DS push ds mov ax,cs mov ds,ax mov ax,250Ch ;revector to our own mov dx,offset read_com ; interrupt handler int 21h pop ds ;restore DS </pre>  |  |        |

**Figure 6:** The assembly language listing for DOSTERM.EXE, a simple interrupt-driven terminal emulation program that allows characters typed on one PC to be sent, via a null modem cable, to the screen of a second PC. DOSTERM performs no handshaking with the external device to which it is linked.

# Monet, not money.



The HP  
PaintJet.  
\$1395.\*

hp HEWLETT  
PACKARD PaintJet

Who says fine art is out of reach? The HP PaintJet color printer produces brilliant color for a price any business can afford.

So now there's no limit to what you can create



with your business communications. Surprise your audience with thousands of colors. Beamed up on an overhead. Or tucked neatly into a report. Persuading people up to 85% more effectively than black and white.

The PaintJet works with all your favorite graphics, presentation, spreadsheet and word processing software. Just hook it up to your IBM-compatible or Macintosh computer and start painting.

For only \$1395 (add \$125 for the Macintosh interface).

Call 1-800-752-0900 Ext. 711K for your nearest authorized HP dealer and a free sample output. The HP PaintJet. It's what artists are starving for.

There is a better way.

 **HEWLETT  
PACKARD**

\*Suggested U.S. list price. Business graphics created using Microsoft® Excel, which is a U.S. registered trademark of Microsoft Corp. ©1989 Hewlett-Packard Company PE12916

**CIRCLE 326 ON READER SERVICE CARD**

```

        assume ds:data
; Initialize the UART to 9600 N81.
;
        mov     ax,00E3h      ;9600 bps, no parity,
        xor     dx,dx         ; 8 data bits, and 1
        int     14h           ; stop bit

; Unmask IRQ4 interrupts in the 8259's IRQ mask register.
;
        in      al,INTA01     ;clear bit 4 to unmask
        and     al,0EFh      ; IRQ4 (COM1) interrupts
        out     INTA01,al

; Initialize the Interrupt Enable Register and assert GPO2.
;
        mov     dx,uart_addr  ;first clear DLAB
        add     dx,3
        in      al,dx
        and     al,07Fh
        out     dx,al

        sub     dx,2          ;set bit 0 for received data
        mov     al,1          ; ready interrupts
        out     dx,al

        add     dx,3          ;assert GPO2, DTR, and RTS
        mov     al,0Bh
        out     dx,al

; Send and receive characters until ESC is pressed.
;
        call    term
; Reset the system and exit.
;
        mov     dx,uart_addr  ;clear bits 0, 1, and 3 of
        add     dx,4          ; the Modem Control Register
        in      al,dx
        and     al,0F4h
        out     dx,al
        sub     dx,3          ;disable UART interrupts
        xor     al,al
        out     dx,al

        in      al,INTA01     ;mask off IRQ4 interrupts
        or      al,10h        ; that reach the 8259A
        out     dx,al

        mov     ax,250Ch      ;reset the int 0Ch vector
        lds     dx,[int0Ch]
        int     21h

        mov     ax,4C00h      ;terminate
        int     21h

main     endp

;;
;; TERM transmits characters typed at the keyboard and displays those
;; received at the serial port.
;;
term     proc    near
; Check the keyboard buffer and process any waiting keycodes.
;
term_loop: mov     ah,1        ;don't read the keyboard if
        int     16h          ; nothing is waiting
        jz      keys_clear

        xor     ah,ah        ;read keycode
        int     16h
        or      al,al        ;ignore extended keycodes
        jz      keys_clear
        cmp     al,01bh      ;exit if ESC was pressed
        jne     output

output:   push     ax          ;display the character
        call    display_char
        pop      ax

        call    send_char    ;output the character

; Check the serial input buffer and read it if a character is waiting.
;
keys_clear: mov     ax,buffer_tail ;loop back if the buffer is
        cmp     ax,buffer_head    ; empty
        je      term_loop

        call    read_char        ;extract character from buffer

        call    display_char     ;display it
        jmp     term_loop        ;return to loop

term     endp

;;
;; READ_CHAR waits for a character to appear in the serial input

```

```

; queue, then reads it and returns it in AL.
;;
read_char     proc    near
no_char:      mov     bx,buffer_tail ;loop until a character
        cmp     bx,buffer_head    ; appears in the serial
        je      no_char          ; input buffer

        cli          ;interrupts off
        mov     al,[bx]          ;read a byte from the buffer
        inc     bx              ; and advance the tail
        cmp     bx,buffer_end     ;wrap around to start of buffer
        jne     read_exit        ; if necessary
        mov     bx,buffer_start
        mov     buffer_tail,bx

read_exit:    sti          ;interrupts on
        ret              ; and exit

read_char     endp

;;
;; DISPLAY_CHAR writes the character in AL to the screen buffer.
;;
display_char   proc    near
        mov     ah,0Eh          ;BIOS TTY function
        xor     bh,bh
        int     10h
        ret

display_char   endp

;;
;; SEND_CHAR writes the character in AL to COM1.
;;
send_char      proc    near
        push    ax              ;save character code
        mov     dx,uart_addr    ;point DX to Line Status
        add     dx,5

send_loop:     in      al,dx      ;loop until Transmit
        test    al,20h          ; Holding register
        jz      send_loop       ; is empty

        sub     dx,5            ;then output the character
        pop     ax
        out     dx,al
        ret

send_char      endp

;;
;; READ_COM handles interrupts generated by COM1 when a byte of data
;; is received. Data is read from the UART's Receive Buffer register
;; and stored in a FIFO queue.
;;
read_com       proc    far
        push    ax              ;save registers
        push    bx
        push    dx
        push    ds

        mov     ax,data         ;establish DS addressability
        mov     ds,ax
        mov     dx,uart_addr    ;make sure DLAB is clear
        add     dx,3
        in      al,dx
        and     al,07Fh
        out     dx,al

        mov     dx,uart_addr    ;read the character
        in      al,dx

        mov     bx,buffer_head  ;calculate next head position
        mov     dx,bx           ; to make sure the buffer
        inc     dx              ; isn't full
        cmp     dx,buffer_end
        jne     no_wrap
        mov     dx,buffer_start
        mov     dx,buffer_tail

no_wrap:      cmp     dx,buffer_tail
        je      exit_int        ;exit if buffer is full

        mov     [bx],al         ;insert character in buffer
        mov     buffer_head,dx  ;advance head pointer

exit_int:     mov     al,20h      ;signal EOI to the 8259
        out     INTA00,al
        sti          ;interrupts on

        pop     ds              ;restore registers
        pop     dx
        pop     bx
        pop     ax

        iret                    ;return from interrupt

read_com      endp

code
ends
end        main

```

## Lab Notes

from the buffer, READ\_CHAR temporarily clears interrupts and keeps them disabled during the critical time interval when it manipulates buffer pointers.

### NEXT UP: OS/2

DOSTERM is about as uncomplicated as a functioning interrupt-driven serial communications program can be. Although it works, it's of limited practical use since it lacks the features we expect from commercial communications packages: the ability to adjust line settings on the fly, to dial phone numbers, to upload and download files, and more. What it does is to illustrate just *how* such programs are implemented under DOS. If you've never seen it done

The DOSTERM  
program illustrates  
how functioning  
interrupt-driven  
serial  
communications  
programs are  
implemented under  
DOS.

before, it may seem a daunting task even to try. Once you know how, you can see that a few cookbook routines are all that you need to get started.

In the next Lab Notes, we'll take a look at the OS/2 communications environment and see how DOSTERM can be adapted to OS/2. You'll find that OS/2's support for serial I/O is much richer, making programs easier to write. Many common communications functions are handled automatically, including input and output stream buffering, RS-232 handshaking, and XON/XOFF flow control. And since standard OS/2 API functions are sufficient to access the driver, programs can be written in C as easily as in assembly language.

Equivalent code listings are thus simpler, shorter, and more concise, which is one indication that the greater portion of the load is being carried by the operating system, not by the application program running on top of it.

More importantly, the OS/2 COM driver works completely in the background and delivers all the functionality needed by commercial-quality programs. The multithreaded programming techniques avail-

able in OS/2 eliminate the code serialization endemic to DOS applications. This shifts the burden of arbitrating between input requests generated at the UART level and output requests generated at the keyboard—for example, from the application to the operating system. It's an exciting prospect for the next issue! ■

Jeff Prosize is a contributing editor of PC Magazine.

## Here's the PC Voice Mail system that can increase your productivity by over 9 weeks.

The average business person **wastes** 5 to 7\* hours each week on the telephone. That's over 9 weeks a year of wasted time and profits. That's why you need Watson.

### What Is Watson?

It's the \$199 hardware and software system that turns your PC and telephone into an intelligent communications system that outperforms voice messaging systems costing thousands of dollars more.

### Why Watson?

Because Watson invented the category of PC voice mail. Because *PC Magazine* selected Watson Version 1 as "Editor's Choice for Product of the Year in 1984." Because Watson Version 6.23 is a Hayes compatible modem (1200 or 2400 BPS) that runs completely in the background without interfering with other computing functions. Because Watson comes with a 60-day free support program. Because Watson has over 30,000 satisfied users. *And because over 45% of our sales come from user referrals!*

### All This For Just \$199.

With basic Watson you'll get a single or multiple user system that answers the phone; forwards messages to any phone, even pagers; provides private and public voice mailboxes; gives you a personal calendar and programmable alarms plus a dictation system with full featured voice editing. You'll get auto dialing, remote access operation and message retrieval. Plus a sortable phone book based on a Rolodex™ file card structure in which you can enter free form notes and do key word searches. And it's all yours for just \$199.

\*George Walther, *Phone Power* (New York: Berkley Books, 1986).

### Watson—Voice Information System (VIS)™ Option.

An English-like command language that allows you to customize voice messages, control message sequences with touch tones for both inbound and outbound response applications.

### Hear All About It.

To decide which Watson is right for you, call our Demo Hotline. You'll hear an actual demonstration and discover all the ways Watson can work for you.

### Call Our Demo Hotline Now.

**1-800-6-WATSON, EXT. 243.** In MA 1-508-651-2186 EXT. 243. Or to order Watson directly, call 1-800-533-6120 EXT. 243 (in MA 1-508-655-6066 EXT. 243).

MasterCard, VISA, and American Express accepted.

### 30 Day Money-Back Guarantee.

Try Watson for 30 days. If you aren't completely satisfied, return it for a full refund.

### FREE Copy Of Phone Power Just For Listening To Our Demo.

We'll send you a free copy of *Phone Power*, if you call before 10/31/89 and ask for extension 243. No order necessary. Over 200 pages of practical techniques for small business owners and company executives. Make your telephone and your time more profitable.



Call on the power of  
**Watson**  
from **Natural MicroSystems** NMS  
8 Erie Drive, Natick, MA 01760-1313

CIRCLE 733 ON READER SERVICE CARD